# Demo: comDeX unveiled demonstrating the future of IoT-Enhanced communities

Nikolaos Papadakis, Georgios Bouloukakis, Kostas Magoutis

# Demo: ComDeX Unveiled Demonstrating the Future of IoT-Enhanced Communities

Nikolaos Papadakis[1,2], Georgios Bouloukakis[2], Kostas Magoutis[1,3]
papadakni@ics.forth.gr,georgios.bouloukakis@telecom-sudparis.eu,magoutis@ics.forth.gr
[1]Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH), Greece
[2]Télécom SudParis, Institut Polytechnique de Paris, France
[3]Computer Science Department, University of Crete, Greece

## ABSTRACT

The rapidly expanding field of **Internet of Things (IoT)** has necessitated the development of effective and efficient systems for handling the vast quantities of data that are generated. However, the inherent diversity and complexity of IoT environments, coupled with the large volume of data, pose significant challenges to achieving interoperability and efficient data exchange. ComDeX [3], is a novel approach designed to meet these challenges. This demo paper presents the prototype of ComDeX, which is designed to facilitate efficient data exchange in smart communities using a federation of message brokers. The prototype harnesses NGSI-LD and MQTT standards along with an advertisement-based mechanism to enable dynamic data exchange. We detail its implementation and key functionalities, demonstrating its applicability through scenarios that mimic real-world smart communities.

## CCS CONCEPTS

• **Information systems** → **Data management systems**; **Information integration**; • **Software and its engineering** → **Message oriented middleware**.

## KEYWORDS

Keywords: IoT, Smart Communities, Data Exchange, Middleware

**ACM Reference Format:**
Nikolaos Papadakis[1,2], Georgios Bouloukakis[2], Kostas Magoutis[1,3]. 2023. Demo: ComDeX Unveiled Demonstrating the Future of IoT-Enhanced Communities. In *The 17th ACM International Conference on Distributed and Event-based Systems (DEBS '23), June 27–30, 2023, Neuchatel, Switzerland.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3583678.3603279

## 1 COMDEX OVERVIEW

In this section, we provide a comprehensive understanding of the ComDeX prototype. Firstly, we delve into the architectural design of ComDeX, discussing its key components and their roles in data exchange. Subsequently, we discuss the implementation details, including the technologies used and the motivation behind their selection. Finally, we explain the process of setting up the ComDeX system, leading us to the demo scenarios in the later sections.


*DEBS '23, June 27–30, 2023, Neuchatel, Switzerland*

ACM ISBN 979-8-4007-0122-1/23/06…$15.00
https://doi.org/10.1145/3583678.3603279

### 1.1 System Architecture

The ComDeX architecture, focusing on property graph-based entities in a federated broker network, ensures scalability across diverse IoT ecosystems. **Entities**, as representations of real-world objects, carry unique identifiers and types in URI format. They are associated with properties and relationships, manipulated through specific API actions in the ComDeX system. **Brokers** manage and distribute entity data within their operation domains. They organize entities in a hierarchical namespace structure, supporting both fine and coarse-grain CRUD actions. The ComDeX's key characteristic is the broker federation, promoting efficient data exchange across communities. Brokers function as data providers or consumers, depending on the context. The advertisement mechanism, implemented via MQTT messages, ensures the visibility of new context information, keeping the network up-to-date. **Action Handlers** act as conduits between publishers/subscribers and brokers, validating API Actions and managing data flow within the system. They direct entity data to the appropriate broker, ensuring data integrity. The ComDeX system architecture is deliberately structured to ensure high scalability. It can seamlessly accommodate the heterogeneity of IoT ecosystems, spanning from straightforward single community setups to intricately woven multi-community networks. We will delve into more detailed explorations of this architecture in the demonstration scenarios that follow.

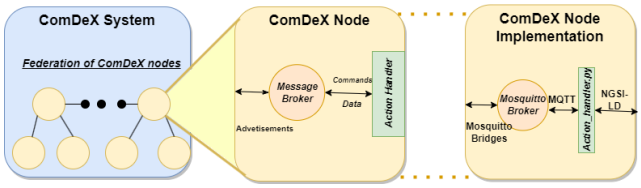


**Figure 1:** ComDeX **high level view and implementation**

### 1.2 Key Implementation Features

We constructed the ComDeX prototype by integrating several technologies to align with the architectural components outlined above. Central to our system is the interaction between Mosquitto, an open-source MQTT broker, (as the MQTT protocol was chosen for its publish-subscribe messaging pattern, small footprint, and resilience to unreliable network environments) and the NGSI-LD [1] information model. Our ComDeX implementation includes a custom Action Handler, written in Python, that bridges the NGSI-LD API and MQTT brokers. While a C implementation may have yielded a more efficient system, Python enabled us to prototype a fully functional federated ComDeX broker in line with the ComDeX design

principles. This prototype can be easily ported to other programming languages. The Action Handler validates incoming NGSI-LD API actions and routes them to the appropriate broker. However, to do this, it must map the NGSI-LD operations, which closely align with HTTP verbs, to appropriate MQTT messages and operations.

*1.2.1 Mapping NGSI-LD Operations to MQTT.* Mapping NGSI-LD operations to MQTT was a complex task, requiring thoughtful design decisions to ensure a seamless integration that preserved data integrity and facilitated efficient data transmission. NGSI-LD data entities are managed via a namespace structured hierarchically as /NGSI-LD/v1/entities/<entity-id>/attrs/<attr-id> which offers endpoints for executing fine-grained or coarse-grained CRUD actions. The following example of a temperature entity with one attribute the current temperature value, illustrates our mapping of NGSI-LD entities to MQTT messages.

```
/* Original NGSI-LD entity */
{
    "id": "urn:ngsi-ld:TemperatureSensor:001",
    "type": "TemperatureSensor",
    "temperature": {
        "type": "Property",
        "value": 25.5,
        "unitCode": "CEL"
    },
    "@context": [
        "https://customcontext.json",
        "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.
             jsonld"
    ]
}

/* Corresponding MQTT Messages */
Topic:smartcity/https%3A%2F%2Fcustomcontext.json/
      TemperatureSensor/ urn:ngsi-ld:TemperatureSensor:001/
      temperature
Payload:
{
    "type": "Property",
    "value": 25.5,
    "unitCode": "CEL"
}

/* Advertisement */
Topic: provider/broker_address/broker_port/smartcity/ https%3A%
      2F%2Fcustomcontext.json/TemperatureSensor/
or
Topic: provider/broker_address/broker_port/smartcity/ https%3A%
      2F%2Fcustomcontext.json/TemperatureSensor/ urn:ngsi-ld:
      TemperatureSensor:001/
```

**Listing 1: NGSI-LD entity message separation**

An NGSI-LD entity maps to several MQTT messages—one for each attribute. In our example, the MQTT topic for the message is `(area+'/entities/'+context+type +id + attribute)`, and the attribute's value is the message's payload. The term "area" (a human-friendly string) defines the broker's geographical scope (e.g., CommunityA_CityB). Each Action Handler-Broker pair should be aware of its covered "area" (smartcity in the example). This knowledge enables data trimming according to a human-friendly area scope. However, the area value is optional and can be omitted. The string "entities" signifies that the message's information belongs to an entity, aiding in prototype development and maintenance. It also helps determine when non-entity-related information is inserted into a broker. As future updates may include context information beyond entities (e.g., active subscriptions), this distinction is important. Each entity as described above consists of four distinct parts, each mapping one-to-one: context, type, id, and attribute. When entity requests are made, suitable messages are amalgamated back into a single unit by utilizing the entity-id to identify messages belonging to the same entity. This process is essentially the reversal of the original data split. Because each attribute corresponds to a unique message, attribute-level operations can be carried out as required. The general objective was to closely adhere to the latest NGSI-LD specification, thereby also supporting pull-based applications, or those that issue GET requests for data, in accordance with the NGSI-LD API. This was achieved by having the data publisher instruct the broker to retain the last message on that topic. Messages can then be deleted as necessary using an empty-payload message. Next, we will delve into the detailed implementation of key NGSI-LD API commands. The creation of an entity takes place using the "POST" entity command to an NGSI-LD broker. The format used is as follows: `'POST'` 'http://broker-address/:port/NGSI-LD/v1/entities/' `'entity_data.file'`. Every possible command selection uses a similar format in the prototype, as exemplified by `python3 actionhandler.py -b broker-address -p port -c POST/entities -f 'entity_data.file'`. The action handler first verifies if the command/action is valid, as is the case with the POST/entities command. It then attempts to connect to the specified broker to validate if the provided entity creation file is a valid JSON, and whether the essential entity parts (id, type, context) are included. Subsequently, it checks if an entity with the specified entity-id already exists in the broker and if so, notifies the publisher. An advertisement, which signals the availability of new context information in a federated ComDeX service, is created for this entity if it does not already exist for this broker. A fresh advertisement is generated and published with the topic "`provider/' + broker_address + broker_port + /area + /entity context + /entity type + (/entity id)`" following the configured advertisement granularity. Finally, a new MQTT message is generated for each distinct attribute present in the entity. To facilitate temporal queries for each attribute, messages for attributes 'createdAt' and 'modifiedAt' are also generated. The process is depicted in pseudocode in Listing 2.

```
# Main Action Handler
def Action_handler(argv):
    # Check if command is POST/entities
    if(command==POST/entities):
        # Connect to the broker
        connect(broker,port)
        # Open and validate the entity data file
        file=open_file("entity_data.file")
        if(is_valid_json(file) and is_valid_ngsild(file)):
            # Check if entity already exists
            if(entity_exists(entity.id)):
                print("Entity_already_exists")
                exit()
            # Check if advertisement exists for the entity
            if(!advert_exists(broker,port,entity.context,entity.type)):
                # Publish advertisement if it doesn't exist
                publish(advertisement)
            # Publish each attribute and its corresponding
                 temporal message
            for each attribute in entity_file:
                publish(attribute.message)
                publish(attribute.time_message)
            # Close the connection
            close_connection()
        else:
            # Print error message for invalid file
            print("Invalid_file")
            exit()
```

**Listing 2: POST/entities command**

The creation of a subscription follows a similar logic to the creation of an entity but with additional intricacies. The action handler first checks if the command and the file provided for entity creation, containing at least one of the following: id, type, watched_attributes, are valid. It then tries to connect with the specified broker to subscribe to the provider messages corresponding to the subscription file. For every provider message/advertisement received for the first time, a new parallel connection/subscription is established with the broker advertising relevant information. If an advertisement is deleted while a subscriber is still connected, the related subscription is terminated. The received data is then reassembled accordingly, and the operation is terminated if a subscription exceeds a timeout. Listing 3 provides the pseudocode of this process.

```
# Main Action Handler
def Action_handler(argv):
    # Check if command is POST/Subscriptions
    if(command==POST/Subscriptions):
        # Open and validate the subscription file
        file=open_file("subscription.ngsild")
        if(is_valid_json(file) and is_valid_ngsild_sub(file)):
            # Subscribe for advertisement notifications
            subscribe_for_advert_notif(broker,port,sub_data)

# Function to subscribe for advertisement notifications
def subscribe_for_advert_notif(broker,port,sub_data):
    # Initialize list of known advertisements
    known_adverts=[]
    # Connect to the broker
    connect(broker,port)
    # Subscribe to advertisements
    subscribe(advertisements)
    # On receiving a message
    on_message():
        # Get advertisement from message topic
        advert=message.topic
        # If message payload is Null
        if(message.payload==Null):
            # Remove advertisement from known adverts
            known_adverts.remove(advert)
            # Close appropriate connection
            close_appropriate_connection()
        # If advertisement is not already known
        if(!advertisement_already_known(message.topic)):
            # Add advertisement to known adverts
            known_adverts.add(advert)
            # Create subscription
            create_sub(advert.broker,advert.port,sub_data)
```

**Listing 3: POST/Subscriptions command**

*1.2.2* ComDeX *Advertisements and Bridging.* Broker federation in ComDeX is primarily grounded on the propagation of advertisements of exported context. These advertisements, akin to entities, are disseminated as MQTT messages. Each advertisement message features a topic in the following format: *(provider/' + broker_address + broker_port + area + NGSI-LD entity context + NGSI-LD entity type).* At the outset of the topic, the string 'provider/' denotes a data-provider advertisement message. The broker address and broker port serve to identify the broker hosting the data as described by the context and type, while the area, represented as a string, indicates the region covered by the data provider. A client seeking specific context information employs these advertisements, available at the broker it initially connects with, to establish new connections and access all desired information. This information could be located at a remote broker. An advertisement may be created for each new entity type or, for a finer granularity, for each new entity id. Our prototype allows users to select the granularity of the advertisement, either by type or id, with per-type currently set as the default. Provider messages are disseminated across the federation of brokers utilizing MQTT bridges. For instance, 'topic provider/# out 2 "" ""' can serve as a bridge, propagating all provider messages generated at the broker to the specified address. The ability to filter data moving between each bridge is a crucial aspect of this bridging. However, caution must be exercised when creating MQTT bridges. Bidirectional connections could result in redundant message transmissions

## 1.3 Setting Up ComDeX

This section provides a step-by-step guide to installing and setting up the ComDeX prototype. The way presented here is manually compiling from the source code. It requires cloning the ComDeX repository from GitHub, which includes all necessary code and configuration files.

(1) Ensure Python 3.7 or later is installed on your system. If not, follow the official Python installation guide[1].
(2) Install Mosquitto MQTT Broker on your system. Follow the official Mosquitto installation guide[2].
(3) Setup Mosquitto with a custom configuration file (mosquitto.config) that is provided in the ComDeX repository.

```
mosquitto -c DEMO/mosquitto.config
```

(4) Clone the ComDeX repository from GitHub.

```
git clone https://github.com/SAMSGBLab/ComDeX.git
```

(5) Install the required Python libraries. These are listed in the 'requirements.txt' file in the ComDeX GitHub repository.

```
pip install -r requirements.txt
```

(6) Navigate to the repository folder and run the 'actionhandler.py' script with the -h flag to get the usage message.

```
python3 actionhandler.py -h
```

After setup, you can ensure that the prototype is functioning correctly by performing a simple subscription and post. Run the following commands at the localhost where the prototype has been installed and using the files in the DEMO/examples/ folder (if the configuration isn't modified just omit the -b broker-address -p port from the CL variables):

```
actionhandler.py -b broker-address -p port \
-c POST/Subscriptions -f 'simple_sub.ngsild'
actionhandler.py -b broker-address -p port \
-c POST/entities -f 'entity_data.ngsild'
```

If the commands execute error-free, the prototype works well.

# 2 COMDEX SCENARIOS

The previous sections have set the stage by detailing the ComDeX system architecture and setup process. Now, we transition into a practical exploration of ComDeX's potential through a series of demonstration scenarios. Each scenario showcases the versatility and robustness of the ComDeX prototype under different community and broker configurations. To ensure our scenarios are grounded in reality, we've chosen the port of Heraklion as our setting. This port exemplifies a diverse IoT ecosystem due to the

[1]https://www.python.org/downloads/
[2]https://mosquitto.org/download/

variety of stakeholders involved in its day-to-day operations. To reproduce or further explore these scenarios, detailed setup, configuration, and sample codes are provided in our GitHub repository[3].

## 2.1 Scenario 1: single community exploiting a central broker

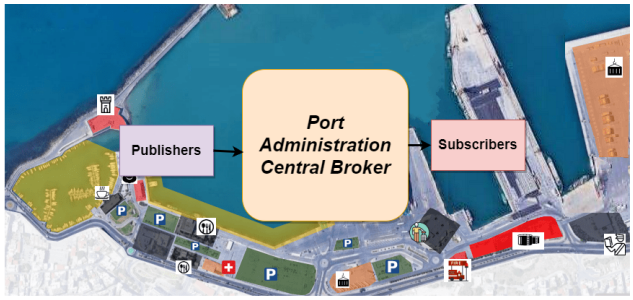


**Figure 2: An example of the port of Heraklion with only one centralised ComDeX node.**

Our exploration commences with a straightforward configuration: a smart port managed by a singular port administration organization. Here, a centralized broker orchestrates all data interactions, offering a clear illustration of ComDeX's application in a single community, single broker scenario. This demonstration provides a foundation for understanding ComDeX's operations and paves the way for more complex configurations. In this contained scenario, all entities are created and consumed at the same broker, allowing us to focus on core operations such as entity creation and subscription-based consumption. While advertisements are created in this context, their function is not essential to this scenario. This scenario also highlights the flexibility of performing context aware operations at various granularity levels, such as using entity id, entity type, or even specific entity attributes.

## 2.2 Scenario 2: single community exploiting a federation of brokers

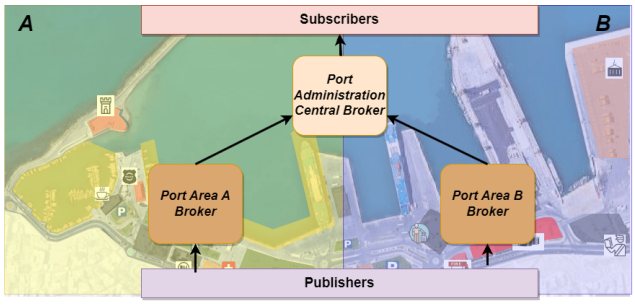


**Figure 3: An example of separating the port of heraklion into a federation of ComDeX nodes based on their location (here into 2 equal locations A and B)**

Building on the principles demonstrated in the first scenario, we delve into a scenario where a single community—the smart port administration organization—is divided among multiple brokers. This scenario explores ComDeX's capabilities in managing data exchange within a more intricate, yet singular, community setup. Here, we demonstrate how a single community can be partitioned into a federation of brokers based on spatial characteristics or application domains (e.g., smart buildings, smart transportation).

[3]https://github.com/SAMSGBLab/ComDeX/DEMO

## 2.3 Scenario 3: multiple communities exploiting a federation of brokers

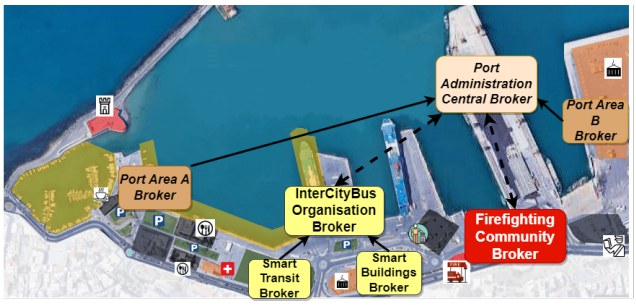


**Figure 4: ComDeX federation deployment within the diverse IoT ecosystem of Heraklion's port.**

Finally, we present a multiple communities scenario to showcase the full spectrum of ComDeX's capabilities. This time in the smart port we consider that there are multiple smart communities involved, such as the firefighting community and the public transportation organization [2]. Each community possesses its own broker network, resulting in a complex, large-scale IoT ecosystem. This scenario underscores the strength of the federated broker network and the advertisement-based mechanism, demonstrating ComDeX's versatility and adaptability. The focus here is on demonstrating how ComDeX facilitates the configuration of multiple communities and how advertisements provide a mechanism for controlling the flow and discovery of information across these ecosystems.

# 3 CONCLUSION

In this demo paper, we have presented ComDeX, a pioneering solution for scalable and efficient data exchange within and across IoT ecosystems. ComDeX's federated broker network, entity-centric data model, and the unique advertisement-based mechanism allow it to thrive in various configurations, from a single community with one broker to complex multi-community scenarios. This work has merely scratched the surface of ComDeX's potential. Future research and development may explore improvements in the prototype implementation. We encourage the community to experience the ComDeX prototype firsthand and share feedback, contributing to the continuous evolution of this promising platform.

# ACKNOWLEDGMENTS

We thankfully acknowledge funding for this research by the Greek RTDI Action "RESEARCH-CREATE-INNOVATE" (ΕΠΑνΕΚ 2014-2020), Grant no. T2ΕΔΚ-02848 (SmartCityBus).

# REFERENCES

[1] 2021. Context Information Management (CIM) NGSI-LD API V1.4.2. https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.04.02_60/gs_cim009v010402p.pdf
[2] Georgios Bouloukakis, Chrysostomos Zeginis, Nikolaos Papadakis, Panagiotis Zervakis, Dimitris Plexousakis, and Kostas Magoutis. 2023. Enabling IoT-Enhanced Transportation Systems Using the NGSI Protocol. In *Proceedings of the 12th International Conference on the Internet of Things* (Delft, Netherlands) *(IoT '22)*. https://doi.org/10.1145/3567445.3567460
[3] Nikolaos Papadakis, Georgios Bouloukakis, and Kostas Magoutis. 2023. ComDeX: A Context-aware Federated Platform for IoT-enhanced Communities. In *The 17th ACM International Conference on Distributed and Event-Based Systems (DEBS)*.